STATE OF AI IN SOFTWARE ENGINEERING

A practitioner-led paper drawing on the collective experience and discussion of members of the CTOs AI Club

Observed period: 9 April 2025 - 15 July 2026

4,066 participant messages | 135 contributors | UK-led CTO community

Central conclusion

AI is already changing the economics and mechanics of software delivery, but the decisive advantage comes from engineering discipline, context design and governance - not from code generation alone.

Written and analysed by Tim Clark | July 2026

Revised August 2026

Foreword

Over the past year, a significant part of my work has increasingly involved understanding not simply what generative AI can do, but what happens when it is introduced into real software engineering and technology organisations.

I have spent more than three decades working in technology, software delivery and transformation, including CTO and senior technology leadership roles. My career has covered the evolution from traditional software engineering and enterprise systems through cloud and digital transformation to the current emergence of generative AI and agentic development.

What makes the present period particularly interesting to me is that AI is beginning to change more than the tools engineers use. It is starting to change the economics, organisation and mechanics of software delivery itself.

In my own work I have been exploring this change from both sides: as a technology leader considering its organisational consequences, and as a hands-on practitioner using AI extensively in the design, development and evolution of software products. That experience has increasingly convinced me that the most important questions are not about which model writes the best code or whether an AI agent can generate an application.

The more important questions are: How should we organise engineering when implementation becomes dramatically cheaper? How do we preserve architectural integrity when agents can make changes across a codebase? How do we verify what has been produced? What happens to engineering skills and career development? And how should CTOs govern systems capable of acting increasingly autonomously?

My own view is that AI will become a normal part of the software engineering environment, but that its greatest impact will not come from replacing engineers with code-generating machines.

Instead, I believe implementation is becoming increasingly abundant while judgement, architecture, context, verification, integration and accountability become more valuable. That is also one of the strongest themes to emerge from the practitioner discussion analysed in this paper. The resulting conclusion is that engineering discipline becomes more important in an AI-enabled organisation, not less.

It was against this background that I became interested in the conversations taking place within the CTOs AI Club.

Unlike much of the public discussion surrounding AI, these conversations were not primarily vendor demonstrations, benchmark comparisons or predictions about an imagined future. They were technology leaders and practitioners exchanging experiences of using these tools in real environments: what worked, what failed, where extraordinary productivity gains appeared, where costs or complexity increased, and what organisational problems emerged as adoption accelerated.

I therefore decided to treat the conversation as a body of qualitative practitioner evidence.

The resulting dataset covers discussions between 9 April 2025 and 15 July 2026 and includes 4,066 messages from 135 contributors, with approximately 700 messages directly associated with software engineering themes.

Rather than reproducing those conversations or attempting to attribute positions to particular individuals, I have sought to identify recurring themes, disagreements, experiences and emerging patterns and organise them into a coherent view of where AI-enabled software engineering appears to be heading.

This is therefore neither an academic study nor a formal industry survey. It is a practitioner-led synthesis of practitioner experience, interpreted through the perspective of someone who has spent much of his career building, operating and transforming technology organisations.

Importantly, the conclusions and interpretations are my own. They should not be regarded as representing the formal position of the CTOs AI Club, RecWorks, its administrators or any individual contributor. The paper deliberately anonymises participants while recognising that the underlying insight belongs to the community of technology leaders who were willing to share their experiences openly.

My hope is that by bringing those experiences together, this paper provides CTOs, engineering leaders, founders and boards with something more useful than another catalogue of AI tools: a practical view of how software engineering itself may need to change.

Tim Clark
CTO & Technology Transformation Leader
July 2026

Acknowledgements

This paper would not have been possible without the contributions of the members of the CTOs AI Club. Over the period covered by this analysis, 135 technology leaders, CTOs, founders, consultants and practitioners contributed more than 4,000 messages to the discussion, sharing experiences, observations, successes, concerns and lessons from the practical adoption of AI in software engineering.

Particular thanks are due to Barry Cranford of RecWorks, the administrator of the CTOs AI Club, for creating and sustaining the community in which these discussions have taken place.

The collective experience of the group forms the foundation of this paper. The analysis, interpretation and conclusions are the author's own and should not be taken as representing the views of the CTOs AI Club, RecWorks, its administrators, or any individual participant. Participant names and identifying information have otherwise been excluded to preserve the private nature of the discussion.

Acknowledgement principle

The practitioner evidence belongs to the community that created it; the synthesis, interpretation and conclusions in this paper are the author's own.

Executive summary

The CTOs AI Club discussion provides a distinctive view of AI adoption in software engineering: experienced technology leaders comparing real tools, real delivery constraints and real organisational consequences. The conversation is neither uniformly enthusiastic nor resistant. It shows a profession moving rapidly from experimentation to operational redesign.

The strongest evidence of value appears in bounded implementation work: generating boilerplate, making contained changes, navigating unfamiliar repositories, drafting tests, reviewing code, accelerating legacy-system modifications and producing prototypes. Several contributors report compressing days or weeks of effort into minutes or days. Yet the same discussion repeatedly records failure modes: incorrect changes, context degradation, security exposure, escalating token cost, merge conflict, weak architecture and teams spending more time configuring tools than shipping.

The emerging pattern is therefore not “AI replaces software engineers”. It is a widening separation between engineers and organisations that can direct, verify and govern AI-generated work and those that cannot. AI amplifies the quality of the system around it. Strong architecture, clear interfaces, automated tests, disciplined review and well-maintained context become more valuable; weak practices scale defects and confusion faster.

The paper concludes that software engineering is entering an agent-assisted phase in which implementation becomes cheaper and more parallel, while specification, system design, verification, integration and accountability become the bottlenecks. CTOs should measure end-to-end delivery outcomes rather than lines of code or self-reported speed, and should treat AI adoption as an operating-model change rather than a tool rollout.

Key findings

1. Adoption has moved beyond autocomplete. The centre of gravity is shifting from code completion to repository-aware agents that plan, edit, test, review and sometimes open pull requests.

2. Productivity gains are real but highly uneven. The largest gains occur with experienced users, familiar stacks, bounded tasks and strong feedback loops.

3. Context is the new engineering constraint. Large codebases, long sessions and undocumented decisions expose model context limits and lead to inconsistent behaviour.

4. Quality does not emerge automatically. AI can produce plausible code faster than teams can understand, test and govern it.

5. Architecture and platform engineering gain importance. Stable APIs, isolated worktrees, automated environments and repository knowledge make parallel agent work more reliable.

6. The workforce effect is polarising, not simply substitutive. Strong engineers become more leveraged; weak review habits and disengagement become more visible and costly.

7. Economics remain unsettled. Model quality, subscription subsidies, token consumption, vendor lock-in and orchestration overhead complicate ROI.

8. Governance must be embedded in delivery. Security, privacy, intellectual property, auditability and production permissions cannot be bolted on after experimentation.

Board-level message

The question is no longer whether engineering teams will use AI. The question is whether the organisation can convert local acceleration into safer, faster and more valuable software outcomes.

1. Scope and methodology

This paper synthesises a private professional community discussion rather than a formal market survey. It is best read as qualitative practitioner research: a structured interpretation of issues raised by CTOs, engineering leaders, founders, consultants and hands-on developers.

Dataset characteristic	Observed value
Observation period	9 April 2025 to 15 July 2026
Parsed participant messages	4,066
Distinct message senders	135
Messages matching software-engineering themes	Approximately 700
Frequently discussed tools and concepts	Claude Code, GitHub Copilot, Cursor, Codex, agentic development, vibe coding, code review, context engineering
Analytical approach	Theme coding, recurrence analysis, chronology and synthesis of practitioner examples

Participant names, telephone numbers and personal identifiers are excluded. Short quotations are anonymised and selected only where they capture a recurring theme. Counts are directional because ordinary discussion uses terms inconsistently and messages can belong to several themes.

Limitations

- The group is self-selecting and is not statistically representative of all software organisations.
- Reported productivity improvements are largely experiential rather than controlled measurements.
- Shared links and claims were not independently validated for this edition.
- The conversation is UK-led but includes experience across sectors, company sizes and technology stacks.

2. From assistant to engineering agent

The discussion traces a clear adoption arc. Early messages focus on Copilot-style assistance, prompt quality and isolated uses such as VBA or Python generation. By mid-2025, attention moves to Cursor, Claude Code and “vibe coding”. By 2026, the emphasis is on agent orchestration, repository context, parallel worktrees, automated reviews, skills, sub-agents and organisational rollout.

Adoption stage	Defining characteristic
Stage 1 - Assistance	Autocomplete, snippets, explanation, documentation and one-off code generation. Human retains the workflow and decides each step.
Stage 2 - Repository collaboration	The model reads project context, modifies several files, runs tests and iterates with the engineer.
Stage 3 - Bounded agency	An agent receives a task or plan, works in an isolated branch or container, and returns a change set for review.
Stage 4 - Multi-agent delivery	Planning, implementation, test, review and documentation are divided among agents, often operating in parallel.
Stage 5 - AI-native engineering system	Context, policy, evaluation, observability, environments and approval gates are managed as shared engineering infrastructure.

“Switched to Claude Code from Cursor ... game changer.”

Anonymised CTO participant, August 2025

The important change is not merely that models write more code. They increasingly occupy parts of the delivery loop that were previously coordinated by humans and tools: interpreting requirements, selecting files, sequencing work, executing commands, creating tests and commenting on pull requests. This makes the agent harness, permissions and context architecture as important as the underlying model.

3. Productivity: substantial, uneven and easy to mismeasure

The chat contains credible accounts of major acceleration. Examples include completing a legacy .NET change in roughly twenty minutes that was expected to take several days; achieving in two days what a different prototyping approach had taken two weeks; and using AI code review to find issues an experienced developer believed they would have missed.

Where gains appear strongest

- Well-bounded changes with an explicit acceptance condition.
- Popular languages and platforms with abundant model training coverage, particularly Python, GitHub, cloud infrastructure and infrastructure-as-code.
- Repository exploration, explanation and onboarding into unfamiliar code.
- Boilerplate, migration work, test generation, documentation and repetitive refactoring.
- Rapid prototypes where learning and validation matter more than production durability.
- Experienced engineers who can decompose work, detect errors and redirect the model quickly.

Where gains evaporate

- Long-running tasks that lose constraints or architectural decisions from context.
- Changes spanning many tightly coupled classes or services.
- Parallel work within weak isolation boundaries, creating merge conflicts and duplicated decisions.
- Unclear requirements, unstable interfaces and incomplete test environments.
- Teams that optimise prompts and tooling while feature delivery stalls.
- Work where generated volume increases review and production churn faster than it increases useful throughput.

Measurement principle

Measure lead time, change failure rate, escaped defects, review time and customer outcome. Do not treat generated code volume or individual perceptions of speed as proof of organisational productivity.

4. Context engineering becomes core infrastructure

A recurring theme is that model capability is constrained by what the system can reliably know at the moment of action. Participants describe long conversations dropping instructions, large repositories exceeding practical context, and agents forgetting earlier architectural decisions. The result can be a convincing local change that violates the wider system.

This shifts engineering effort toward context engineering: deciding which specifications, architectural records, interfaces, examples, policies and repository facts should be available; keeping them current; and presenting them in a form that agents can retrieve and apply.

Minimum viable context layer

- Concise repository-level instructions and engineering principles.
- Machine-readable build, test and deployment commands.
- Current architecture and decision records, not abandoned documentation.
- Stable service contracts and explicit ownership boundaries.
- Examples of accepted implementation patterns and known anti-patterns.
- Task-specific context retrieval rather than endlessly expanding one conversation.
- A clear definition of "done", including tests, security checks and review evidence.

"Keep the conversation short ... otherwise it drops context and starts to misbehave."

Anonymised engineering leader, March 2026

5. Quality, security and technical debt

The discussion rejects two simplistic positions: that AI inherently creates technical debt, and that AI-generated code is inherently high quality. The more useful interpretation is that AI accelerates the consequences of the surrounding engineering system. In a disciplined environment it can apply patterns, generate tests and identify defects at scale. In a weak environment it can create weeks of plausible but poorly understood code in a day.

Failure mode	What it looks like	Control
Plausible correctness	Code compiles or passes the happy path while violating edge cases, non-functional requirements or domain rules.	Explicit acceptance tests, adversarial cases and domain-owner review.
Permission overreach	Agents execute destructive commands, push changes or modify production-connected resources.	Least privilege, isolated environments, protected branches and approval gates.
Security opacity	Non-specialists can ship complete products without recognising critical vulnerabilities.	SAST/DAST, dependency scanning, secrets detection, threat modelling and security sign-off.
Review overload	More generated changes create a queue of shallow approvals and "review monkey" behaviour.	Smaller change sets, risk-based review, AI-assisted review plus accountable human ownership.
Architectural drift	Each task is locally reasonable but collectively inconsistent.	Architecture tests, contract checks, decision records and designated technical stewardship.
Knowledge erosion	Teams cease to understand code they operate.	Explain-before-merge, rotation, incident ownership and periodic human-led design reviews.

"Vibe coding allows someone without the required security expertise to get further than before."

Anonymised CTO participant, August 2025

6. Architecture and platform engineering are more valuable, not less

As implementation becomes cheaper, the value of boundaries and enabling infrastructure rises. Participants report that parallel agent work is effective when tasks can be isolated behind stable API contracts and separate worktrees; it is much less effective when multiple agents edit the same service and productivity is lost resolving conflicts.

The emerging AI-enabled platform therefore includes more than a model licence. It combines reproducible environments, repository indexing, policy, identity, observability, test automation, cost controls, secure secrets handling and standard agent workflows.

Design principles for an agent-ready codebase

- Prefer modular systems with explicit contracts and ownership.
- Make tests fast, deterministic and runnable without privileged production access.
- Treat documentation and architecture decisions as executable context assets.
- Provide isolated sandboxes or ephemeral environments for autonomous work.
- Constrain agent actions through policy-as-code and protected delivery paths.
- Log prompts, tool calls, changes and approvals sufficiently for audit and learning.
- Optimise for small, reversible changes rather than large autonomous rewrites.

7. The engineering workforce: leverage, polarisation and a new apprenticeship problem

The conversation does not support a simple forecast of wholesale developer replacement. It does, however, anticipate smaller teams, higher expectations per engineer and a widening gulf between people who use AI with judgement and those who either reject it or delegate without understanding.

Experienced engineers often gain the most because they possess the mental models required to decompose work, notice omissions and evaluate trade-offs. This creates a challenge for junior development: traditional learning depended on writing routine code, receiving review and gradually building system intuition. If routine implementation is delegated too early, organisations may weaken the pipeline that produces future senior engineers.

Capabilities that become more important

- Problem framing and requirement clarification.
- System design, interface definition and trade-off reasoning.
- Test design, verification and evidence-based review.
- Debugging across model output, code, infrastructure and business process.
- Security and operational risk judgement.
- Context curation and agent workflow design.
- Communication with product, operations, compliance and boards.

Talent implication

Do not remove the junior pathway; redesign it. Junior engineers should use AI, but remain accountable for explaining changes, designing tests, handling incidents and demonstrating system understanding.

8. Tool economics, vendor dependence and ROI

Participants frequently compare subscriptions, token limits and model quality. The discussion suggests that cost cannot be separated from task success: a cheaper model that requires repeated correction can be more expensive than a premium model that crosses the required quality threshold. Equally, premium agents can consume credits rapidly on large contexts or poorly bounded tasks.

There is also a two-layer dependency: the underlying model and the agent harness tuned around it. Switching models may degrade the workflow even when benchmark performance looks similar. Current pricing may also reflect strategic subsidy, so business cases should not assume today's unit economics are permanent.

A practical ROI equation

Net AI engineering value

Value of earlier delivery + avoided engineering effort + quality improvements - licence and token cost - review/rework - platform and governance cost - incident and lock-in risk.

The most reliable business case starts with a small number of repeatable engineering journeys and compares end-to-end outcomes against a baseline. Broad mandates such as "all developers must use AI" create activity but not necessarily return.

9. Governance as part of the developer experience

Security, privacy, intellectual property, compliance and audit concerns appear throughout the chat. The appropriate response is not to prohibit useful tools by default, nor to permit uncontrolled experimentation. Governance must be embedded into the engineering workflow so that the safe path is also the easiest path.

Governance control	Expected practice
Approved capability tiers	Define which models and agents may be used for public, internal, confidential and regulated workloads.
Data handling	Prevent secrets, personal data and restricted source code from entering unapproved services.
Identity and permissions	Give agents named identities, least-privilege access and environment-specific controls.
Human accountability	Every merged or deployed change has an accountable human owner, regardless of generation method.
Evidence and audit	Retain sufficient records of agent actions, tests, reviews and exceptions.
Evaluation	Test tools against representative internal tasks, including security, reliability and cost - not only public benchmarks.
Incident response	Define how to stop agents, revoke access, recover repositories and investigate AI-assisted changes.

10. The emerging AI-enabled engineering operating model

The evidence points to an operating model in which people set direction and remain accountable while agents perform bounded, observable work. The objective is not maximum autonomy. It is the highest useful autonomy consistent with risk, reversibility and organisational understanding.

Lifecycle stage	Human-AI division of responsibility
Discover	Human-led problem framing; AI supports research, repository exploration and option generation.
Specify	Human defines outcome, constraints and acceptance evidence; AI helps refine stories, edge cases and implementation plans.

Lifecycle stage	Human-AI division of responsibility
Design	Human owns architecture and trade-offs; AI challenges assumptions and drafts artefacts.
Build	Agent implements bounded slices in isolated environments using approved tools and context.
Verify	Automated tests and AI review run first; accountable engineers assess risk, behaviour and maintainability.
Release	Policy gates, protected branches and progressive delivery limit blast radius.
Learn	Delivery metrics, defects, agent traces and cost data improve prompts, context and platform controls.

11. Recommendations for CTOs and engineering leaders

1. Start with delivery bottlenecks, not tool enthusiasm. Select use cases where implementation, review, legacy comprehension or repetitive change is demonstrably constraining value.

2. Establish a baseline before rollout. Capture lead time, review effort, defect escape, deployment frequency, recovery time and engineering cost for comparable work.

3. Create an approved agent platform. Provide secure models, repository access, sandboxes, logging, secrets controls and standard workflows so teams do not assemble unsafe shadow stacks.

4. Invest in context quality. Maintain concise repository instructions, architecture records, interface definitions and runnable tests. Assign ownership for context assets.

5. Bound autonomy by risk. Allow greater autonomy for reversible low-risk tasks; require tighter oversight for data, security, payments, regulated decisions and production infrastructure.

6. Redesign code review. Review intent, evidence, risk and architecture - not merely syntax. Keep generated changes small and require accountable human approval.

7. Train for judgement, not prompting tricks. Develop decomposition, verification, security, systems thinking and communication. Tool-specific prompting will age quickly.

8. Protect the learning pipeline. Ensure junior engineers still diagnose faults, explain designs, write critical tests and participate in incidents.

9. Track cost at task level. Measure subscriptions, tokens, retries, review time and rework per completed outcome. Reassess vendor economics regularly.

10. Build organisational learning loops. Record successful patterns, common failures, model regressions and exceptions; feed them into shared guidance and evaluations.

12. A 90-day adoption roadmap

Period	Priority actions
Days 1-30: Baseline and control	Map current AI use; identify sensitive repositories; approve initial tools; define policy, ownership and baseline metrics; select two or three bounded use cases.
Days 31-60: Instrumented pilots	Run pilots with trained teams; capture task-level cost and outcomes; implement sandboxing, logs, automated checks and review templates; compare against baseline.
Days 61-90: Scale what works	Standardise proven workflows; retire ineffective

Period	Priority actions
	experiments; expand repository context; create an internal evaluation set; publish operating guidance and a quarterly value/risk review.

13. Conclusion

The state of AI in software engineering is best described as operationally significant but institutionally immature. The tools are already capable of compressing meaningful engineering work, navigating legacy systems and extending the reach of skilled individuals. They are also capable of scaling weak assumptions, insecure defaults and shallow understanding at unprecedented speed.

The winning organisations will not be those that generate the most code or deploy the most agents. They will be those that redesign software delivery around clear intent, high-quality context, strong architecture, automated evidence, bounded autonomy and human accountability. In that model, AI does not remove engineering discipline; it makes discipline the primary source of competitive advantage.

Final takeaway

Implementation is becoming abundant. Reliable understanding, integration and accountability remain scarce.

Appendix A - Indicative theme signals in the discussion

The following counts are keyword-based directional signals and are not mutually exclusive. They show the relative intensity of recurring topics in the source discussion.

Theme	Messages matched	Interpretation
Jobs, skills and engineering roles	412	High recurring attention
Models and tools	371	High recurring attention
Quality, defects, testing and risk	326	High recurring attention
Cost and return on investment	323	High recurring attention
Agentic systems and agents	248	Material recurring attention
Coding assistants and vibe coding	201	Material recurring attention
Governance, privacy and compliance	192	Material recurring attention
Productivity and delivery speed	151	Focused recurring attention
Architecture, context and codebases	147	Focused recurring attention

Appendix B - Practitioner propositions for further validation

- AI coding delivers the highest net value when work is bounded, testable and reversible.
- The productivity distribution between engineers widens after AI adoption.
- Context quality predicts agent success more reliably than model benchmark position alone.
- Agent-enabled teams require fewer implementers but more architecture, platform and verification capacity.

- Organisations that remove junior roles without redesigning apprenticeship will create a future senior-skills deficit.
- AI-generated change volume can worsen end-to-end flow unless review and integration are redesigned.
- Task-level economics will matter more than licence price as agent workflows mature.

Source and acknowledgement note: This paper is derived from the CTOs AI Club WhatsApp discussion and reflects the collective practitioner experience shared by its contributors. The author gratefully acknowledges the members whose observations, experiences and debate made this analysis possible, and Barry Cranford of RecWorks for administering and supporting the CTOs AI Club community. The synthesis, interpretation and conclusions are the author's own and do not represent an official position of the CTOs AI Club, RecWorks or any individual participant. Direct quotations are anonymised and lightly edited for brevity.